

Performance Analysis of a New Packet Trace Compressor based on TCP Flow Clustering

Raimir Holanda, Javier Verdu, Jorge García and Mateo Valero
Computer Architecture Dept. - Technical University of Catalonia
{rholanda,jverdu,jorge,mateo}@ac.upc.edu

Abstract

In this paper we study the properties of a new packet trace compression method based on clustering of TCP flows. With our proposed method, the compression ratio that we achieve is around 3%, reducing the file size, for instance, from 100 MB to 3 MB. Although this specification defines a lossy compressed data format, it preserves important statistical properties present into original trace. In order to validate the method, memory performance studies were done with the Radix Tree algorithm executing a trace generated by our method. To give support to these studies, measurements were taken of memory access and cache miss ratio. For the time, the results have showed that our proposed method provides a good solution for packet trace compression.

1. Introduction

Packet traces of Internet traffic are important tools for performance evaluation and design purposes of many network elements. For instance, packet traces can be used to evaluate prototypes of network systems such as routers, firewalls, Internet servers, etc. Packet traces can also be used to feed trace-driven simulators used in the performance evaluation and design of many components of these systems, as for example, the Network Processors performing functions ranging from basic packet forwarding to quality of service (QoS) processing, packet classification, security, billing and accounting.

The performance of these systems depends not only on parameters such as packet length or inter-packet time, but also on some properties of flows, that we call *semantic properties*: spatial and temporal locality of IP address, IP address structure, and TCP flags sequence. It is essential, thus, to use in these studies packet traces which maintain representative semantic properties of the Internet traffic.

Despite the importance of using correct packet traces in the performance evaluation studies, there are several

reasons that make difficult in many cases to have access to the required traces. Firstly, Internet providers are usually reluctant to make public real traces captured in their networks. When these traffic traces are made public, they are delivered after some transformations, such as sanitization, which modify some basic semantic properties (such as IP address structure). Others problems arise due to the increasing speed of Internet routers. Hardware for collecting traces at high speed (e.g. to link rates of 2.5 Gbps, 10 Gbps or even 40 Gbps) is usually expensive. Moreover, with the increase of link rates, the required storage for packet traces of meaningful duration becomes too large.

As an example, let us consider the problem of storing a 1000 sec trace taken from a link at 10 Gbps. Storing the full content of the traffic would require an storage of 1.25 Tbytes. If we only store the 40 bytes TCP/IP headers, together with timing information, we would require a storage capacity of 125 Gbytes (assuming a mean packet length of 400 bytes).

In this paper we address the problem of the compression of these potentially huge packet traces, assuming the more common case of storing the TCP/IP packet headers plus timing information only.

Content compress can be as simple as removing all extra space characters, inserting a single repeat character to indicate a string of repeated characters, and substituting smaller bit strings for frequently occurring characters. The compression is performed by algorithms which determine how to compress and decompress. Some of the most popular compress algorithms are the Huffman coding [1], LZ77 [2], and deflate [3]. Those specifications define lossless compressed data formats. From our measurements, using these methods we can expect a compression ratio of around 50%.

The previous methods do not take into account the specific properties of the data to be compressed. A more effective compression method is the one proposed by Van Jacobson [4] in the context of transmission of In-

ternet traffic through low speed serial links. The method is based on the fact that in TCP connections, the content of many TCP/IP header fields of consecutive packets of a flow can be usually predicted. As we will show, the achievable compression ratio using this method is around 30%, reducing the file size, for instance from 100 MB to 30 MB. In the context of saving storage space of potentially huge packet traces, a lossy method that utilizes the flow nature in Internet traffic to reduce data volume while preserving some informations for network research is presented in [5]. In this case, headers packet traces are reduced to 16% of its original size.

In this paper we study the properties of a new web packet trace compression method proposed in [6]. The method is lossy, in the sense that cannot recover the exact original packet trace. Exploiting some properties of Internet Web packet flows the compression ratio we achieve is around 3%, reducing the file size, for instance from 100 MB to 3 MB.

We have restricted our studies to Web traffic, which is an important part of the current Internet traffic. Other applications, such as P2P applications, are becoming more and more important, and we plan to incorporate them in future studies.

2. Flow characterization

In [6] a novel flow characterization that incorporates a specific set of packet characteristics such as TCP structures, inter packet time, and payload size, was proposed. This flow characterization can be used for achieving a lossy compression method. In this section we summarize the main ideas behind this flow characterization and compression method.

Let us define a packet flow as a sequence of packets in which each packet has the same value for a 5-tuple of source and destination IP address, protocol number, and source and destination port number. Each packet flow can be characterized using the following set of parameters: TCP flags, inter packet time into a flow, and payload size.

Let

$$P^m = (P_1^m, P_2^m, \dots, P_m^m) \quad (1)$$

denote a flow of m packets, where P_i^m is the i -th packet of this flow. For P_i^m we define a mapping M , $M(P_i^m) = F_i^m$, where F_i^m is an integer. For each flow, let

$$F^m = (F_1^m, F_2^m, \dots, F_m^m) \quad (2)$$

denote a vector of m integers and $G^m = \{ F^m \}$ be a set

of these vectors. The mapping M is a function given by:

$$F_i^m = M(P_i^m) = \sum_{j=1}^3 W_j X_j(P_i^m) \quad (3)$$

where (W_j) are different weights to give a relative importance to each parameter. Below, we define each one of the $X_j(P_i^m)$ parameters:

- $X_1(P_i^m)$ represents the TCP flag carried by each packet P_i^m :

$$\begin{aligned} X_1(P_i^m) &= 0; \text{if } syn = 1, ack = 0 \\ &= 1; \text{if } syn = 0, ack = 1, fin = 0 \\ &= 2; \text{if } rst = 1 \\ &= 3; \text{if } syn = 0, ack = 1, fin = 1 \end{aligned}$$

others TCP flags arrangement can be found, but we have restricted our studies for the most common;

- $X_2(P_i^m)$ represents the inter packet time into a flow. If a packet to be transmitted waits for a packet sent by the opposite node, it is called a dependent packet, otherwise, if a packet is sent immediately after the last one we classify it as not dependent. For instance, in the TCP three-way handshake, when a node sends a *syn* TCP flag, it waits for a *syn+ack* TCP flag from the opposite node. This waiting time corresponds to the Round trip time (RTT). In this sense, we associate inter packet time to acknowledgment dependence. Hence, it is reasonable to believe that with a known RTT distribution and for small flows might be possible to model the inter packet into each flow using only a parameter: the acknowledgment dependence. Hence:

$$\begin{aligned} X_2(P_i^m) &= 0; \text{if } ack_dependent_packet \\ &= 1; \text{if } NOT_ack_dependent_packet \end{aligned}$$

- $X_3(P_i^m)$ represents the packet size:

$$\begin{aligned} X_3(P_i^m) &= 0; \text{if } packetsize \leq 60Bytes \\ &= 1; \text{if } 61 < packetsize \leq 1400 \\ &= 2; \text{if } packetsize > 1400 \end{aligned}$$

Furthermore, we have used different weights (W_j) . In our case, we have used the following weights for TCP flags (W_1) , acknowledgment dependence (W_2) , and payload size (W_3) : $W_1=16$, $W_2=4$, and $W_3=1$. Evidently, depending on the type of problem to be studied, we can apply different weights. Hence, the W_j weights give us a higher degree of flexibility.

2.1. Web flow clustering

We have used the flow characterization previously described to study the diversity of Web flows in a high-speed link. We have concluded that Web flows are not very different from each other, and many of them have identical or very similar F_i^m values. More extensive results are presented in [6].

The flow diversity study among Web flows is based on clustering techniques [7]. Clustering techniques allow the grouping of many elements of a set in different groups, clusters, whose members are as similar as possible.

The clustering methodology starts from a real trace, and after isolating the flows by their number of packets, we generate for each flow a corresponding F^m vector.

We have applied our methodology to traces from different available packet traces [8] [9]. The main conclusion is that in consequence of the huge similarity among Web flows, we can group a high amount of them into few clusters.

3. Packet trace compression

After the analysis presented in [6], we see that 98 percent of the flows have less than 51 packets. These flows comprise 75 percent of all Web packets transmitted on the link and 80 percent of the bytes on average. Similar traffic characteristics were found for instance, in [10], [11]. Hence, our compression method takes into account two types of flows: *short* and *long flows*. *Short flows* are flows with the amount of packets ranging from 2 to 50, and *long flows* are flows with more than 50 packets.

Our compressor generates four datasets. A first dataset called *short-flows-template* stores the templates of flows with less than 51 packets. This dataset has a first field that stores the value of m (number of packets), and then a sequence of F_i^m values. The second dataset is called *long-flows-template*. It stores the templates of flows with more than 50 packets. The first field stores the value m and then, for m packets, the F_i^m value and the inter packet time. The third dataset, *address*, stores a sequence of unique IP destination address found in the trace. Finally, the fourth dataset, *time-seq* stores for each flow, the time-stamp of the first packet. Furthermore, it stores the following fields: a dataset identifier (S=short-flow-template, L=long-flow-template), an index to a specific template position into the template dataset, the RTT of short flows and another index to the *address* dataset. In the case of short flows, we have seen that time-varying does not represents a serious problem.

Hence, for short flows, we have assumed that each flow has a specific RTT. Evidently, this assumption is not true for long flows, where RTT is dynamic and time-varying. Hence, for long flows, the field RTT in the *time-seq* dataset is not filled, and the inter packet time is stored in the *long-flows-template* dataset.

Our compression method works finding F^m vectors that are repeated or very similar. The method starts looking into the 5-tuple of fields (source and destination IP address, source and destination port number, and protocol number). When a packet carrying a new flow is found, a new node is inserted at the end of a linked list and a new record is created in the *time-seq* dataset. Each node stores the following fields: a key (a hashing of source and destination IP addresses, source and destination port numbers, and protocol number), time-stamp, F_i^m value and two pointers. Each node has associated another linked list, where are inserted the packets from the same flow. When a *Fin* or *Rst* TCP flag is found, the algorithm, first of all, looks for the number of inserted nodes associated to this flow.

In the case of short flows, the algorithm searches for identical or similar F^m vectors in the *short-flows-template* dataset. In the case that a match is not possible, we insert a new record in this dataset, update the *time-seq* dataset and remove all nodes of this flow from the linked list. This new F^m vector will constitute the center of a new cluster. In the case of a match, we only update the *time-seq* dataset and remove all nodes of this flow from the linked list. According with the flow characterization described on Section 2 and for the same i , the maximum distance between two F_i^m values of different flows is 50. Consequently, for flows with m packets, the maximum inter flow distance is $(m \times 50)$. We have assumed that two vectors r and s are similar whether the difference among them is lower than 2% of the maximum inter flow distance. Therefore, d_{rs} is given by:

$$d_{rs} \leq (m \times 50 \times 0.02) \quad (4)$$

For long flows, we do not perform any search because the probability of find two identical F^m vectors is really very low. Hence, each long flow generates an input in the *long-flows-template* dataset. This approach does not influence the performance of the method, because the number of long flows is limited. The most important realization is that shorter flows are much more common than longer flows, so we have paid attention to compressing the short flows fast.

4. Decompression algorithm

The decompression algorithm sets up a linked list to store temporarily the sequence of decompressed packets. It works reading the four compressed datasets: *time-seq*, *short-flows-template*, *long-flows-template* and *address*. As we have seen, the *time-seq* dataset stores, for each flow, the time-stamp of the first packet; the *short-flows-template* dataset stores the templates for small flows; the *long-flows-template* dataset the templates for long flows and the *address* dataset, a sequence of unique IP destination addresses. Those templates store the necessary information to reproduce important packet flow characteristics such as: inter packet time, TCP flag sequence and packet size. The algorithm starts reading the *time-seq* dataset. Note that this dataset is sorted by the time-stamp data field. Reading the dataset identifier and template position fields, the algorithm identifies the template file to be read (*short-flows-template* or *long-flows-template*) and places the position of the template in this file. Furthermore, it reads the IP destination address and, in the case of short flows, the RTT.

The algorithm goes reading the sequences of F_i^m values and decoding the TCP flag, the payload size, and the inter-packet time. For each one of the F_i^m values, the algorithm inserts a new node at the linked list sorting by the time-stamp. Furthermore, are assigned the source and destination IP address, source and destination port number and protocol number. For source address, we assign randomly an IP class B or C address. At the moment, as we are working only with Web traffic, we have assigned a random value between 1024 and 65000 to client port number, and to the server side the value 80.

After reading the last F_i^m value of the template, the algorithm continues the procedure reading the next record in the *time-seq* dataset. At this moment, all nodes in the linked list with time stamp less than the current value are written in a decompressed file.

5. Compression ratio

To study the efficiency of the proposed compression method, we compare, for large packet traces, the compression ratio of different compression methods. The measures were taken from a TSH (Time Sequence Header) header trace file and the compression methods evaluated were the GZIP [12], the Van Jacobson method [4], the Peuhkuri method [5], and the method proposed in [6]. The *GZIP* application and also *ZIP* and *ZLIB* [13] use the deflation algorithm. For different TSH file sizes, the compressed file size obtained using the GZIP appli-

cation is 50% of the original TSH file size (see Figure 1).

For the Van Jacobson method we must modify slightly the original method. We assume that a time stamp (2 bytes) is added to each header. The number of active flows can be much more larger in a high-speed Internet link than in a low speed serial link (the scenario which Van Jacobson was originally proposed). We must increase thus the number of bytes needed to store the flow identifier (from 1 byte to 3 bytes). The TCP checksum, however, is not included. As a result we assume that minimal encoded headers are of 6 bytes.

To estimate the compression ratio for the Jacobson method we must use flow-length distribution measured in the available packet traces. We will call P_n as the probability that a web flow has n packets. With the changes we have explained before, the compression ratio for n -packet flows using the Van Jacobson method is bounded by:

$$f^{VJ}(n) = \frac{40 + 6(n - 1)}{40n}, \quad (5)$$

obtaining thus a compression ratio given by:

$$C_{Ratio}^{VJ} = \sum_n P_n f^{VJ}(n) \quad (6)$$

Using the distribution P_n that we obtained from several traces, we conclude that the compression rate of the Van Jacobson method is around 30%.

The lossy method proposed by Peuhkuri also utilizes the flow nature in Internet traffic, but was thought for reduce storage space. However, it has the compression ratio bounded by 16%.

In the proposed compression method 8 bytes are sufficient to represent each flow of n packets. There are some data structures with information related to the clusters of flows that are also needed. However these additional data structures are almost constant with the packet trace length. Then for large packet traces, the compression ratio for n -packet flows is given by:

$$f(n) = \frac{8}{40n}. \quad (7)$$

obtaining thus a compression ratio of:

$$C_{Ratio} = \sum_n P_n f(n) \quad (8)$$

which results in a compression ratio of around 3%.

Figure 1 shows the file size comparison between the original trace and the four compression methods under analysis.

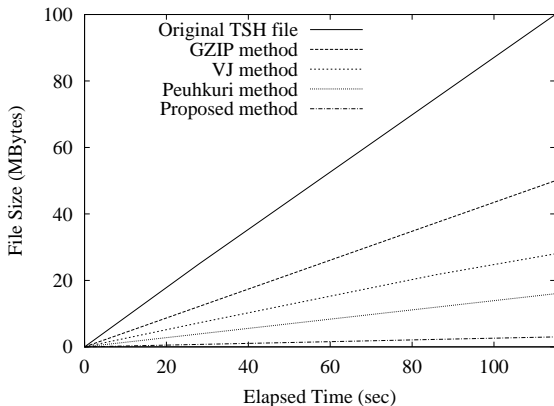


Figure 1: File size comparison

6. Memory performance validation

The compression method studied in this paper achieves a large compression ratio. However, it is not able to recover the exact compressed information. In this section we study whether the recovered trace is suitable for studies focusing on memory access characteristics. The results presented in this section do not cover the entire set of possible network benchmarks, but they clearly show that the recovered trace exhibits close behavior in relation with the original packet trace.

We have applied three benchmark programs taken from Netbench [14] and Commbench [15] benchmarks. The selected programs were: Route (Netbench), NAT (Netbench), and RTR (Commbench). All the selected programs involve the Radix Tree Routing inside their algorithms. The Radix Tree is a binary tree, which starting at the root, stores the prefix address and mask so far. As you move down the tree, more bits are matched going one way down the tree. If they don't match, the other branch holds the entry required. This sort of data structure can result in efficient average performance for forwarding table lookup times, on the order of \ln (number of entries), which for large routing tables is quite a gain. The returned value from looking up an entry will typically be the next hop IP router.

The Radix Tree code was instrumented using the ATOM tool [16]. In order to delimit the processing of packets, checkpoints were placed at the beginning and at the end of the packet processing. The instrumented code records the number of memory accesses performed by each packet. At the end of the traffic trace processing, a list including the total number of memory accesses per packet is generated.

6.1. Memory access measurements

In our experiments, we have used four different traces. A first trace is a subset of the original RedIRIS trace, containing only Web flows. Henceforth, we will call this trace of Original trace. The second one is the decompressed trace, obtained by applying our proposed compress/decompress method over the Original trace. A third trace was generated assigning random IP destinations addresses, but maintaining the same temporal distribution of the Original trace. Finally, for the last trace, the IP directions were generated by a multiplicative process and were launched using LRU stack model with an exponential inter-packet time distribution.

Figure 2 plots the cumulative traffic (Y axis) against the number of memory access (X axis) when executing the Radix Tree Routing algorithm for the four traces. We observe that the Original and the Decompressed trace show similar behavior while the others traces depict different shapes. We can see, for instance, that approximately 55% of the traffic from the Original and Decompressed trace execute access to memory ranging from 53 to 67. Otherwise, the random trace shows that only 30% of its traffic ranges from 53 to 62 access to memory, and the fractal trace for this same number of memory access presents approximately 27% of the traffic. Furthermore, we observe that for the Original and Decompressed trace, the number of access ranging from 53 to 92 corresponds to 60%, but for the Random trace, we have 70% of the traffic executing from 53 to 88 memory accesses, and the random trace executing from 53 to 96 accesses to memory for 37% of the traffic. These divergences are due to the fact that the number of visited nodes is different.

6.2. Cache miss rate

In Figure 3 and for the same Radix Tree algorithm, we show the cumulative traffic (Y axis) against the cache miss rate (X axis). Here, again, we observe huge similarity among the Original and the Decompressed trace, but in this case, the fractal trace has a similar behavior and the random trace presenting not concordance with the Original trace. In the graph we can see that about 60% of the packets from the Original and Decompressed trace show a cache miss rate lower than 5%, which correspond to the sequence of packets with a very similar behavior. Otherwise, for this same ratio, we obtain around 10% of the packets from the Random trace. For a cache miss ratio ranging from 5% to 10%, we observe an inverse behavior, with 50% of the packets from the random trace conforming this ratio and only 10% of the packets from

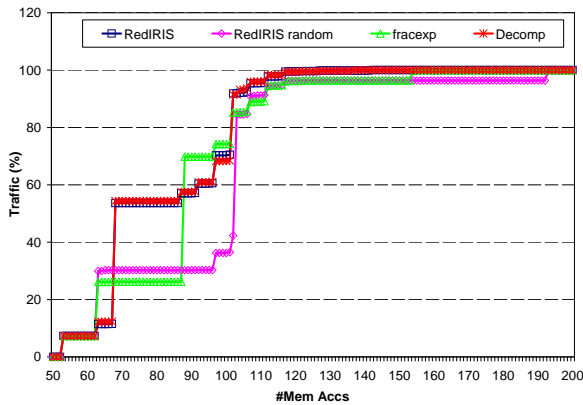


Figure 2: Memory Access for the traces: original RedIRIS, the random address, the fractal address, and the decompressed trace generated by our method.

the Original and Decompressed trace. In our opinion, the differences between the Original and random trace are due to the fact that in one trace memory needs to be released, whereas in the other trace memory is still available.

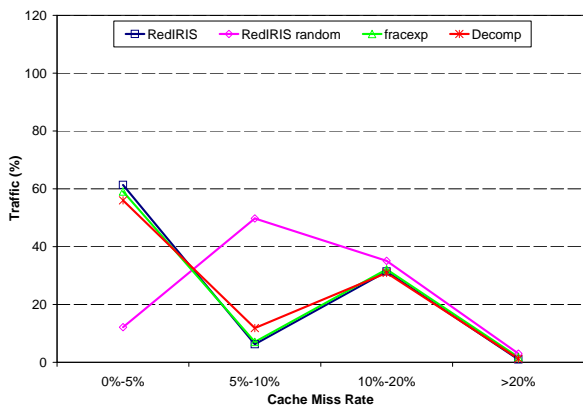


Figure 3: Cache miss rate for the traces: original RedIRIS, the random address, the fractal address, and the decompressed trace generated by our method.

7. Conclusions

In this paper, we have studied a novel packet trace compression method based on TCP flow clustering. We show that with our proposed method, storage size requirements for TCP/IP packet traces are reduced to 3% of its original size. Although this specification defines

a lossy compression method, analysis over the decompressed trace have showed that they represent a good approximation of original traces. A memory performance evaluation was carried out with four types of traces and the outcomes for memory access and cache miss ratio measurements demonstrated that our proposed compression method shows a huge efficiency.

As future works, we intend to extend this analysis to a richer set of network benchmarks, verifying also the applicability of the method to other types of applications like P2P, and implement a synthetic packet trace generator based on the described methodology.

Acknowledgments

This work was supported by CAPES-Brazil, by the Ministry of Science and Technology of Spain under contracts TEC-2004-06437-C05-05, TIC-2001-0995-C02-01 and TIN-2004-07739-C02-01, under grants CIRIT 2001-SGR-00226, BES-2002-2660, VI FP project EuroNGI and the HiPEAC European Network of Excellence.

References

- [1] D. E. Knuth. Dynamic Huffman coding. In *Journal of Algorithms*, 6:163-180, June, 1985.
- [2] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. In *IEEE Transactions on Information Theory*, Vol. 23, n 3, pp. 337-343.
- [3] DEFLATE. Compressed data format specification. In Available in <ftp://ds.internic.net/rfc/rfc1951.txt>.
- [4] Van Jacobson. Compressing TCP/IP headers. In *RFC 1144*, February, 1990.
- [5] M. Peuhkuri. A method to compress and anonymize packet traces. In *Proceedings of the ACM SIGCOMM Internet Measurement Conference*, November, 2001.
- [6] R. Holanda, J. Garcia, and V. Almeida. Flow Clustering: a New Approach to Semantic Traffic Characterization. In *12th Conference on Measuring, Modelling, and Evaluation of Computer and Communication Systems*, Dresden Germany, September 2004.
- [7] R. Jain. The Art of Computer Systems Performance Analysis. In *John Wiley Sons, Inc., New York*, 1991.
- [8] RedIRIS. Spanish National Research Network. In <http://www.rediris.es>.
- [9] NLANR. Measurement and Network Analysis. In <http://moat.nlanr.net>.

- [10] L. Guo and I. Matta. The War Between Mice and Elephants. In *Technical Report BU-CS-2001-005, Boston University, Computer Science Department*, May, 2001.
- [11] N. Brownlee and K. Claffy. Understanding Internet traffic streams: Dragonflies and tortoises. In *IEEE Communications Magazine*, 40(10):110–117, October, 2002.
- [12] J. -L. Gailly and M. Adler. GZIP documentation and sources. In *ftp://prep.ai.mit.edu/pub/gnu/*.
- [13] J. -L. Gailly and M. Adler. ZLIB documentation and sources. In *ftp://ftp.uu.net/pub/archiving/zip/doc/*.
- [14] Gokhan Memik, William H. Mangione-Smith, and Wendong Hu. Netbench: A benchmarking suite for network processors. In *IEEE International Conference Computer-Aided Design (ICCAD)*, 2001.
- [15] Tilman Wolf and Mark A. Franklin. CommBench - a telecommunications benchmark for network processors. In *Proc. of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, , June 1994.
- [16] Amitabh Srivastava and Alan Eustace. ATOM - A system for building customized program analysis tool. In *Programming Language Design and Implementation (PLDI), pages 196-205*, June 1994.